Automatic code generation for ATLAS communications drivers

M. Vivó, M. Fairén and À. Vinacua

Department of Software, U.P.C.
Diagonal 647, 8^{ena} planta
E08028 Barcelona, Spain

December 9, 1999

Abstract

ATLAS is a software development platform created in our Department. Among other benefits, it provides support to easily distribute applications over a network. In these applications, communications issues among the different processes should be faced. Pursuing to isolate application developers from the intricacies of these issues, communication drivers are automatically generated from an interface declaration of each process. This automatic code generation –not unlike the generation of stubs in CORBA [1, 2] from the IDL specification– is the main topic of this report.

1 Introduction

ATLAS is a software development platform designed and implemented in our Department with a twofold objective:

- To transparently offer a collection of services to applications developed over it and
- To facilitate the integration and reuse of different components developed in the Department.

Among the services provided under the first of these is included the possibility of breaking up the application into modules that can be distributed between nodes of a LAN. This gives rise to many different problems that need to be addressed [3], and the emphasis in ATLAS's design is to relief the programmer from most of them. We have made great efforts to make the use of the platform as easy and transparent as possible.

ATLAS is an evolution of a system described in [4]. It inherits some aspects of the architecture of the previous one, and adds robust and fault-tolerant network distribution transparently, a meta-journaling system, a much more flexible

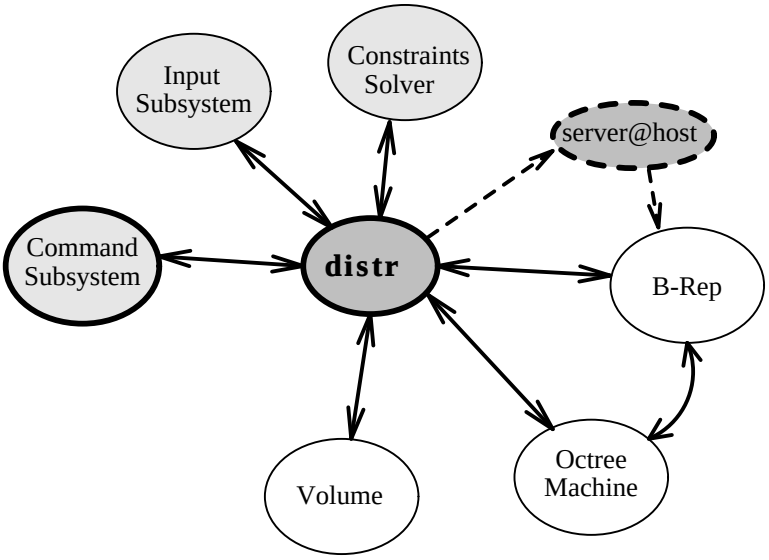

Figure 1: A sample execution of an ATLAS application.

control language and an orthogonal design which affords much more flexibility to the applications built in it.

In this report we will emphasize the ATLAS approach to achieve a total transparency to the developer with respect to the mechanisms related to the processes communications. To this end, we shall widely explain the way the ATLAS communications drivers work and how they are automatically generated by ATLAS in order to hide all of their internal intricacies to the developer.

1.1 Overview of the ATLAS architecture

The ATLAS architecture is represented in figure 1, where the ovals denote processes and the arrows represent communications between them. It is a centralized architecture where the process `distr` acts as the master process and is the center of each ATLAS application. This architecture allows to implement easily an intelligent distribution of processes. The `distr` process decides the processes distribution dynamically depending on availability, load and aptitude of each host in the network to run each application process.

The ATLAS communications mechanism is implemented using sockets, and its implementation uses the wrapper classes for sockets offered by the public domain package ACE_Wrappers (see [5] and [6]).

Figure 1 shows a typical ATLAS application. The processes belonging to the ATLAS platform are shaded, whereas white balloons denote user application processes. The processes depicted with a thick line represent the main components of ATLAS. All other processes are regarded equally by the system, and they don't need to know about each other.

Of these three main processes, the most crucial is the master process `distr`. This is the process that the user starts up to invoke the application. It acts as a communications center for the duration of the execution, and provides some essential services to all other modules. It is also responsible of the *journaling* mechanism –which allows replays of sessions and supports undo's and redo's– enforcing also the consistency of data recorded in the *journal* after editions or modifications of it; the *fault-tolerance* of the system, recovering processes after disruptions of communications or failures of individual processes; and the dispatching mechanism that assigns an input datum, provided by an input system, to the corresponding request, normally issued by another process.

The process `server` is a daemon that runs on each host configured to run ATLAS applications in the network. A user can select a specific list of hosts via variables in his environment, or else ATLAS attempts to use all resources configured by the administrator (depending on their load). A broadcast message sent by `distr` periodically is answered by the `server` of each host which includes in this answer information about the name of the host, the architecture, the current load and the list of processes that this host is able to execute. All this information is kept by `distr`, and each time a new process needs to be loaded and connected to the rest of the application, `distr` looks up this information in order to decide which is the best host to execute this process at that moment and connects to the `server` on this target machine requesting that such a process be started for him. `server` then forks a copy of itself, makes the appropriate verifications, loads the adequate environment and execs the desired process. Figure 1 only shows one such `server` for legibility, although there will be a server running on every node available to ATLAS.

The third of the ATLAS main processes is the `command subsystem` which guides the application behavior by interpreting programs and instructions written in a language (ATL) designed for ATLAS and described in [7]. ATL is a powerful control language which allows the developer to describe his application and aspects of its user interface (see section 3.1) and also allows the final user to introduce his own macros. ATL is a modular language where a module is a file written in this language which can be:

- a description of the interface of the corresponding ATLAS process, including also the definition of commands adequate to that process,
- a definition of some commands useful for the application, but without being directly related to any ATLAS process definition; e.g. commands defining the interaction among several processes.

1.2 Motivation

Since ATLAS' first priority is to offer the maximum transparency to the developer, the design of its architecture must hide the intricacies of the interprocess communications from the programmer.

The communications mechanism has to address problems like how to start a process in an application, how to manage the interchange of information between

processes and also how to detect failures in the application's communications in order to know when the *fault-recovery* mechanism should be activated.

The interchange of information between processes has also an added difficulty when they are running in an heterogeneous network because data can be interpreted with different meaning depending on the architecture where they are used.

The problem of actually transferring the data robustly has long since been solved. Indeed we just rely on XDR[8] for that purpose, because it is a standard representation known by most of the different architectures. However other problems remain. ATLAS wants to hide from the application developer everything related to this encapsulation of data and its conversion from and to the corresponding XDR representation, making this mechanism as transparent as possible.

Moreover the ATLAS process communications require quite a bit of code in each process devoted to handshaking with `distr`, generating the *heartbeat* messages at the adequate rate, preparing the arguments for process routines or collecting results and encoding them for being transported over the network, and dispatching calls to processes routines. But the programmer should be relieved from these tasks.

2 Communications driver

An ATLAS application process execution is based on a *remote-procedure-call* paradigm. A process can be then considered as a set of routines to do the process related work plus a communications driver to manage the interchange of messages with the rest of the application.

The communications driver is the main program of the process, and its role is to listen to requests or messages from the master process and other connections added and dispatch them as needed. The most frequent requests or messages sent by the master process are routine calls, data answering a request, or an ATLAS event notification.

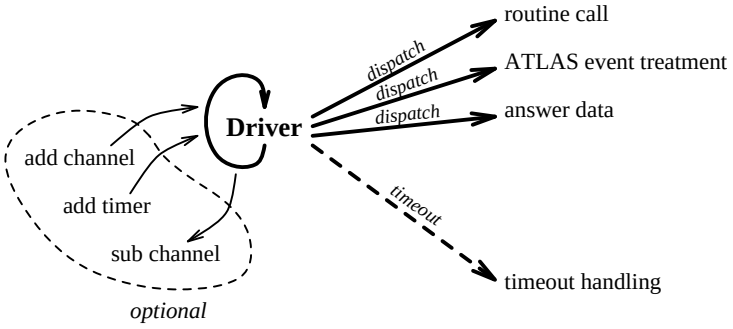

Figure 2: Driver role scheme.

The execution of an ATLAS process is based then on the "Dispatch" method of its communications driver. It enters in a loop listening to the group of channels of this process (the default is only the connection with the `distr` process). When a message arrives, it serves this message and keeps on listening for the next one (see figure 2). Apart from receiving and serving messages the driver can also pay attention to interruptions (like signals).

The default for an ATLAS process driver is to manage the channel connecting with the master process and serving the interruption of SIGALRM, which is used in the *heartbeat* mechanism. Users may add other channels for special purposes (for example to listen to X-windows events).

The *heartbeat* mechanism makes every process being executed in the application send a short message periodically to the master process giving the required information to control the global status of the execution. This mechanism detects if a process or the communication with it fails, therefore making it necessary to activate the *fault-recovery* mechanism.

But the actual driver is much more flexible (as can be seen in figure 2). It also allows the process to add channels to listen to, add a timer treatment or remove channels added before. This flexibility only requires to have implemented the treatment for messages being received by these new channels.

Aside from routine calls, parameters and answers to requests –see section 4–, a process may receive other kinds of messages from `distr`, including:

- The contents of some input data item that the user introduced and is still in the system's internal structures. This message is produced in response to an explicit request of this information made by the process,
- an order to finish the process execution. This order causes the call to a finishing routine that the developer can define specially for each process,
- a system event notification. The ATLAS events mechanism gives the opportunity that an application process be informed about changes in ATLAS internal structures (what processes are in execution, if there is some request waiting for an input data, etc). The process asks for a subscription to a particular ATLAS event and `distr` sends this event notification to it when this event is produced.

There are also other special messages a driver may receive, but they are only used by the Command Subsystem (a command, a return value or a return parameter). Although it is not advisable to change the Command Subsystem process (because it is a heavy-weight component of ATLAS), if some application intended to replace the Command Subsystem, it would have the possibility to use these special messages for its communications.

3 About the ATL language

3.1 Defining the process interface

The process interface is a module written in ATL language which defines the prototype of the public routines of the process and the needed types for their parameters and return results.

The ATL language (described in detail in [7]) is an imperative and modular language designed for ATLAS applications. It allows the definition of types, variables, functions and procedures that can be exported (visible to other modules) or local. It also accepts the most common control structures inside functions and procedures (conditionals, loops, etc.) and routine calls both synchronous and asynchronous.

Although only the prototypes of its remotely invocable routines and its types are needed for a process interface, the module defining the process interface can also include functions or procedures defined in ATL which describe the interaction with other processes in the application or with the user (e.g. asking for input data). ATL modules which are not the interface of any process but define the execution coordination and interaction between processes can also be defined in the application, and users may dynamically add their own. This possibility allows configurability of the application execution at run time, rapid prototyping by recompiling ATL modules also at run time and offers a macro-definition language to the end user.

```
USE se;
EXPORT #deftype point STRUCT
                        x -> real;
                        y -> real;
                        z -> real;
                      ENDSTRUCT
EXPORT #deftype simplex STRUCT
                          name -> string;
                          vertices -> VECTOR [4] OF point;
                        ENDSTRUCT
EXPORT #deftype scene VECTOR [100] OF simplex
EXPORT #deftype property integer

EXPORT scene totalsc;
       ...
PROT
  EXTERN PROCEDURE segmentation (scene &sc, property p);
  EXTERN PROCEDURE display_scene (scene sc);
  EXTERN FUNCTION contained_in (point p) RETURNS simplex;
     ...
ENDPROT
      ...
EXPORT PROCEDURE SegmentSimplex () IS
  segmentation (totalsc, GETDATA("Input the property value"));
  display_scene (totalsc);
  se::Sortida ("Segmentation completed","m");
ENDPROCEDURE
```

Figure 3: Portion of the interface definition in ATLAS for the volume modeling process ("volum").

An example of an ATL module with part of the interface of a process called `volum` can be seen in figure 3. This example is not complete, but it shows the definition of a set of types exported by the module (some of them are needed

as parameters of external routines), the prototypes of two external routines (these prototypes and the types of its parameters would form the interface of the process), and the description of a procedure (being also exported to make it visible to other modules) that combines the execution of the process routines, asks for an input datum (through *GETDATA*) and also calls a procedure of another module (`se::Sortida`).

3.2 The code generator and the compiler grammar of the Command Subsystem

The ATLAS process communications require quite a bit of code in each process devoted to handshaking with `distr`, generating the *heartbeat* messages at the adequate rate, preparing the arguments for process routines or collecting results and encoding them for being transported over the network, and dispatching calls to process routines. To handle this, ATLAS automatically generates code stubs that the developer must link with his program. These stubs are constructed from the interface declaration of the process (like in figure 3), which contains the type definitions used for variables to be exported and the prototype definitions of the process external routines.

Since the same language (ATL) is used to define the process interface (section 3.1) and also to describe the application behaviour (interpreted at run time by the Command Subsystem –see section 1), we use the same grammar to parse the file in both cases: when the generator is going to generate the code stubs for the communications mechanism (section 4) and when the interpreter of the Command Subsystem is interpreting the ATL code at run time. The difference between the two is determined by the value of a flag that indicates whether the parser is doing a code generation or it is directly interpreting code (run time).

When the parser is doing a code generation (which is the main subject of this report), it is mainly interested in extracting information from the exported type definitions and from the prototypes of extern functions or procedures, because this is the relevant information to build the code stubs for the communications driver.

4 Automatic code generator

4.1 Overview

We can distinguish between two main goals that the automatically generated driver solves. On one hand, we have the *heartbeat* mechanism –explained in section 2–, used by ATLAS in order to detect communication failures with processes. On the other hand, calling remote routines implies handling data interchanges between processes, both for parameters and results.

The automatically generated code addresses both objectives. The generator creates the driver's code in three files (explained in detail in section 4.7) that are eventually linked with the developer's code.

In this section we will talk in depth about the design decisions these requirements lead to, and we will show how the automatically generated code works.

4.2 Calls, parameters and results

The distributed environment proposed by ATLAS follows a remote routine call paradigm. Processes are waiting for requests to execute their routines. In ATLAS, it makes no difference whether these requests come from the user, through the command system, or from another process (in fact, the command system is itself another process).

In the '.atl' file defining its interface, each process offers (makes public) some types and routines. These routines are of one of the two following kinds. The first one is composed by those routines implemented in the ATL module itself, written in ATL language. Routines in the second category are external routines, implemented in *C++*, by the process developer. They are called external routines, because they are executed by the ATLAS process and not by the ATLAS Virtual Machine. Once running, each process will be waiting for requests coming from the system to execute its external routines. So, the need for communication between the system and the process arises: calls and parameters should be passed from `distr` to the process and results should go the other way (see figure 1).

For each external routine call, `distr` will send the involved process a message communicating the request, and one additional message for each parameter of the routine. After executing that routine, the process will send back a message with the return value (that may be void) and an additional message for each parameter passed by reference.

4.3 Bridge types

In order to make type control checking, types of the parameters of the external routines declared in the '.atl' file must be defined (or imported) there. As these are ATL type definitions and not *C++* types, conversions must be made somewhere from the internal ATLAS storage of variables to *C++* variables.

ATLAS handles internally all variables using a general wrapper class called *Variable* (see [9]). Objects of this class are passed through the network encoded in an XDR stream. The driver will receive this XDR messages, extract the *Variable*s from it, and convert them into appropriate *C++* variables.

The automatic code generator produces code that carries out the conversion from the XDR message to a *bridge C++* type. The communications driver is able to recover the *Variable* from the XDR message. On the other hand, the bridge class definition and conversions from and to it are provided in auxiliary routines (see section 4.4).

In figure 4 we can see an example of an ATL type and its corresponding *C++* bridge type. All the information needed to construct this *C++* class is provided by the ATL definition of the type, and therefore all the code for the

```
ATL type definition:

    EXPORT #deftype point STRUCT
                            x -> real;
                            y -> real;
                            z -> real;
                          ENDSTRUCT

Bridge type:

    struct atl_point {float x; float y; float z;
    atl_point() {}
    atl_point(Variable &v) {
       if (v.Arbre()==NULL) atl_exit(-1); // Invalid variable
       x = ((nodereal *)(*(v.Arbre())).accedir(0))->Getvalor();
       y = ((nodereal *)(*(v.Arbre())).accedir(1))->Getvalor();
       z = ((nodereal *)(*(v.Arbre())).accedir(2))->Getvalor();
       }
    operator Variable() {
       Type t("volum::point","S(x real,y real,z real)");
       Variable v(t,"");
       v.crea_arbre();
       *((*(v.Arbre())).accedir(0)) = x;
       *((*(v.Arbre())).accedir(1)) = y;
       *((*(v.Arbre())).accedir(2)) = z;
       return (v);
       }
       };
```

Figure 4: Conversion from an ATL type to the corresponding bridge type.

bridge type shown in figure 4 has been automatically generated. As can be seen, the conversion from and to an ATLAS *Variable* has been automatically generated. The first direction is provided with a constructor of the bridge class. The other one is made using a conversion operator.

With this mechanism, the conversion from the received XDR message to the bridge *C++* type is made transparently to the developer, as also is the conversion from a bridge type to a *Variable*, a step needed in order to send back results and parameters of the call passed by reference.

So, the only step where the process developer must act is in the last conversion: from the bridge type to the actual parameter type in the *C++* side. This last step is kept manual in order to offer more flexibility: ATL types need not exactly fit the definitions of the application's *C++* classes. Furthermore, incompatibilities between *C++* and ATL types can be circumvented.

In counterpart, the developer must write the routines to convert an object from and to the bridge type. The automatically generated code will use this conversions to translate parameters and results. It should be noted that this conversion is usually very simple and easy to write, as the bridge class members may match the actual class ones. This two-step design has been adopted so that developers need not handle *Variables* directly in most situations. A scheme of the different representations a parameter has in its "trip" through the network can be seen in figure 5.

4.4 Auxiliary routines

In order to isolate application developers from communications topics, an auxiliary routine is generated for each external routine. The auxiliary routine carries

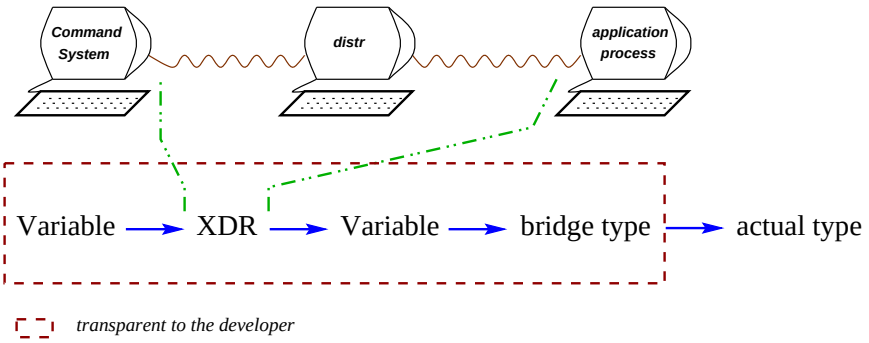

Figure 5: Scheme of the different parameter representations along its trip through the network.

out all the type conversions, from and to the XDR representations that travel through the network and to the actual *C++* classes of the parameters and return value of the external routine. This is done using the bridge types explained in the previous section. Figure 5 shows one way of this conversions. The other way is necessary in order to send return values and pass-by-reference parameters back to the external routine requester.

From the ATL declaration of an external routine, the automatic code generator extracts the information on the type and number of parameters and the type of the return value (see section 4.2).

When an ATLAS process requests the execution of an external routine from another ATLAS process, `distr` receives this request. As many requests for the same routine may arrive at a time, an identifier has to be assigned to each of them in order to be able to send the return value and the existing pass-by-reference parameters back to the appropriate process. So, `distr` assigns each call a different identifier code.

An auxiliary routine receives two parameters. The first one is the code identifying the call. This code will be included in the return messages the auxiliary routine will send to `distr`. The second parameter consists of the list of actual parameters of the external call. These parameters will be translated to the appropriate *C++* types through the bridge types and passed to the process routine for which the auxiliary routine acts as an interface.

When the actual execution of the process routine finishes, the auxiliary routine sends a message for each pass-by-reference parameter back to `distr`. An additional message is sent for the return value of the routine. If this return value is of type void, a special message is sent to indicate the end of the execution. In this last situation, if there exists any pass-by-reference parameters, the return-void message is not necessary (as the parameter itself is enough to indicate the end of the execution), and thus it is not sent.

An example is shown in figure 6. In this example the process routine receives a parameter passed by reference and has no return value, so only the parameter is sent back to `distr` at the end of the process routine. As has been pointed out

```
1    void aux_segmentation(String codi,DLList<Variable *> &parametres) {
2       Pix p=parametres.first();
3       atl_scene ptp0(*(parametres(p)));
4       scene par0(ptp0);
5       parametres.next(p);
6       property ptp1(((nodeenter *)parametres(p)->Arbre())->Getvalor());
7       parametres.next(p);
8       segmentation(par0,ptp1);
9       ptp0=par0.conversio_a_tipus_pont();
10      Variable *rp0=new Variable(ptp0);
11      ReturnParam *retpar0=new ReturnParam(codi,rp0);
12      distrib.envia(retpar0);
13      }
```

Figure 6: Example of an auxiliary routine.

before, the routine receives two parameters, the code identifying the call and the parameter list. Line 3 contains the conversion from the *Variable* representation to the bridge class. The conversion from the bridge class to the actual parameter class happens in the next line. The actual routine is called in line 8. The pass-by-reference parameter is translated to the bridge type in line 9 and to *Variable* in the next one. The last two lines construct the message and send it back to `distr`. The message contains the identifier of the call and the parameter itself. A more complete example can be seen in section 5.

4.5 Management of remote procedure calls in an ATLAS process

In order to carry out the dispatch of calls to the routines offered by an ATLAS process (its external routines), some more code is needed in the automatically generated code. The missing piece is a structure bridging between messages and the auxiliary routines introduced in the previous section. This structure is provided by the class *gestio_crida_a_rutina* (that is, routine call management). The class definition is shown in figure 7. One instance of this class is created in the automatically generated code for each process. It is in charge of receiving and storing routine calls and parameters and of calling auxiliary routines.

Let's take a closer look at this structure. Basically, it consists of two maps: each external routine is linked with its auxiliary routine (through the class member *crida_a*) and each external call is linked with its parameter list (through the class member *crides_pendents*). The first link is established in the initialization step of the process, inside the automatically generated procedure *ini_per_crides* (an example is shown in section 5). The second link is created each time that a request for a routine reaches the process. Notice that an external routine cannot be linked with a unique parameter list, because many calls to that routine may be arriving at the same time.

When a routine call message sent by `distr` arrives to the process driver, the method *nou_missatge_crida* (that is, new call message) is invoked. The parameters of the method are the code identifying the call (see section 4.4), the name of the routine and the number of parameters of the routine (this information is extracted from the routine call message). If the number of parameters is zero, the corresponding auxiliary routine is instantaneously invoked. Otherwise, the

```
typedef void (*rut_tract_crida)(String,DLList<Variable *> &);

class gestio_crida_a_rutina {
    VHMap<String,rut_tract_crida> crida_a;    // table to store the routines to be
                                              // called for each function
    VHMap<String,dades_crida *> crides_pendents;  // table to store the routines and
                                                  // parameters while they are arriving
  public:
    gestio_crida_a_rutina() : crida_a((rut_tract_crida)NULL,20),
                              crides_pendents((dades_crida *)NULL,20) { }
    void lligar_nom_crida(String nom,rut_tract_crida rut) {crida_a[nom]=rut;}
    void nou_missatge_crida(String codi,String nom,int npars) {
       if (npars==0) crida_a[nom](codi,DLList<Variable *>());
       else {
          dades_crida *aux = new dades_crida(nom,npars);
          crides_pendents[codi]=aux;
          }
       }
    void nou_missatge_param(String codi,Variable *v) {
       crides_pendents[codi]->afegir_parametre(v);
       if (crides_pendents[codi]->ja_tots_els_parametres()) {
          crida_a[crides_pendents[codi]->nom()] (codi,
                                      crides_pendents[codi]->obtenir_parametres());
          crides_pendents[codi]->alliberar_parametres();
          delete crides_pendents[codi];
          crides_pendents.del(codi);
          }
       }
  };
```

Figure 7: Class *gestio_crida_a_rutina*

call is stored and the auxiliary routine will be effectively invoked when all the parameters of the call have arrived.

When a parameter message sent by `distr` arrives to the process driver, the method *nou_missatge_param* (that is, new parameter message) is invoked. It adds the parameter to the call's parameter list. If all the parameters of the involved call have arrived, the call to the auxiliary routine is done. Notice that the parameters of an external call arrive in order, because they are sent in order by `distr`.

```
class dades_crida {
      String funcio;                  // function name
      DLList<Variable *> parametres;  // pointers to the received parameters
      int quants_parametres_falten;   // how many parameters are missing
   public:
      dades_crida() : funcio(""),quants_parametres_falten(0) {}
      dades_crida(String n,int np) : funcio(n),quants_parametres_falten(np) {}
      String nom() {return funcio;}
      DLList<Variable *> &obtenir_parametres() {return parametres;}
      bool ja_tots_els_parametres() {return (quants_parametres_falten==0);}
      void afegir_parametre(Variable *v) {
         parametres.append(v);
         quants_parametres_falten--;
         }
      void alliberar_parametres() {
         for (Pix p=parametres.first();p!=0;parametres.next(p))
            delete parametres(p);
         parametres.clear();
         }
   };
```

Figure 8: Class *dades_crida*

An auxiliary class is used to temporally store the parameters of the calls. This structure, *dades_crida* (that is, call data), is shown in figure 8. For each call,

it stores the name of the routine, its parameter list and how many parameters are still missing.

4.6 *main()* part of processes

The code generator also constructs the *main()* module of the ATLAS process. It consists of the code needed to control the network communications plus code for the auxiliary routine introduced in subsection 4.4.

Part of this code is the same for every ATLAS process. So, the generator simply appends the process-dependent code with the process-independent part.

```
#pragma implementation "taula.h"
#pragma implementation "Map.h"
#pragma implementation "VHMap.h"
#include "globals.H"
#include "ComunicDistr.H"
#include "Driver.H"
#include "String.h"
#include "gestio_crides.H"
#include "Variable.H"
#include "DLList.h"

Comunic_Distr distrib(CANAL_COMUNIC_DISTR);
String nomprogram;
gestio_crida_a_rutina gestor_crides_ext;
Driver driv(distrib);
```

Figure 9: Process-independent code: (*head*)

```
void main(int argc,char **argv) {
   nomprogram=argv[0];
   ini_per_crides();
   driv.set_name_program(nomprogram);
   ini_process();
   driv.Dispatch();
   close(CANAL_COMUNIC_DISTR);
   exit(0);
   }
```

Figure 10: Process-independent code: (*tail*)

Figures 9 and 10 show the two pieces of code that remain the same for every process. As can be seen, they define and use global variables that are in charge of communications and external routine call management:

- The *distrib* object encapsulates the communication channel with `distr` and is responsible of the dispatching of messages depending on its contents. The description for this *Comunic_Distr* class is:

```
class Comunic_Distr : public MyEventHandler
{
    ACE_SOCK_Stream canal_com;
    Receiver_socket rebuts;
    FILE *fd_stream;
    typfunc Events[NEVENTS+1];  // to keep the callbacks for ATLAS events
    bool waitsemph, newimalive;
    ImAlive im;             // This message cannot be created dynamically because
                            // it is used in the interruption call.
    bool oob_dos_bytes;     // flag to bypass the Out_Of_Band Data error
                            // detected in Solaris 2.5
```

```
    void soc_viu ();
    void Handle_Message (Message *miss);

  public:
    Comunic_Distr (ACE_HANDLE fd);
    void ini_event_function (int ev, typfunc f);
    void initialize (char *c) { oob_dos_bytes = (c[0]=='T'); }
    bool is_oob_dos_bytes () { return oob_dos_bytes; }
    int envia_oob ();
    int envia (Message *miss);    // method to send a message to distr

    ACE_HANDLE get_handle () const { return canal_com.get_handle(); }
    int handle_input (ACE_HANDLE fd);   // method called when there is input
                                        // through this channel
    int handle_signal (int signum, siginfo_t * = 0, ucontext_t * = 0);
};
```

- The *driv* object encapsulates the process driver itself, it manages the loop listening on the process' channels (at least the channel communicating it with `distr`). It also offers the possibility to add and remove other channels to be listened on. The description for this *Driver* class is shown below.

```
class Driver
{
    ACE_Reactor reactor;
    String nomprogram;

  public:
    Driver (Comunic_Distr &d);
    Driver (Comunic_Comp &d, String nom)
      {
        nomprogram = nom;
        reactor.register_handler (&d, ACE_Event_Handler::RWE_MASK);
      }
    void set_name_program (String nom) { nomprogram = nom; }
    void Add_handler (ACE_Event_Handler &e,
                      ACE_Reactor_Mask mask=ACE_Event_Handler::RWE_MASK)
      { reactor.register_handler (&e, mask); }
    void Add_handler (int signum, ACE_Event_Handler *new_sh,
                      ACE_Sig_Action *new_disp=0, ACE_Event_Handler **old_sh=0,
                      ACE_Sig_Action *old_disp=0)
      { reactor.register_handler (signum, new_sh, new_disp, old_sh, old_disp); }
    void Remove_handler (ACE_Event_Handler &e,
                         ACE_Reactor_Mask mask=ACE_Event_Handler::RWE_MASK)
      { reactor.remove_handler (&e, mask); }
    void Add_timer (ACE_Event_Handler &e, const ACE_Time_Value &delta,
                    const ACE_Time_Value &interval, const void *a=NULL)
      { reactor.schedule_timer (&e,a,delta,interval); }
    void Dispatch () { for (;;) reactor.handle_events (); }
};
```

- The *gestor_crides_ext* object has been explained in subsection 4.5

It is also worth mentioning that the *ini_process* routine called in the *main* function (figure 10) is here in order to allow the developer to make some initializations of the process before entering the dispatching loop. This routine does nothing by default, but the developer can redefine it to include the process initializations at this point.

The process-dependent code simply consists of the auxiliary routines and the procedure *ini_per_crides*. The task of this procedure is to initialize the structure that links external routines with their corresponding auxiliary routines (see subsection 4.5). An example is shown in section 5.

4.7 Automatically generated files

The names of the files containing the automatically generated code consist of the name of the process (procname) prefixed by *atl_*, and the extension depends on each file. These are:

atl*_procname.*hh This file defines the *C++* prototypes for the process routines declared as external routines in the ATL module. It also may include the header file implemented by the process developer (`procname.h`) and includes the generated `atl_procname.H` file.

atl*_procname.*H This file has the *bridge types* implementation for those types used by the external routines of the process (exported types in the ATL module). It includes the `Variable.H` file needed for the conversion of *bridge types* and those generated "`.H`" files corresponding to the modules used by it (in the example of section 5 the `volum.atl` module uses the `se` module so the `atl_volum.H` generated file includes also the `atl_se.H` file.

atl*_procname.*C This file implements the main code for the communications driver and also the auxiliary routines introduced in section 4.4.

5 Putting everything together: an example

5.1 Automatically generated code

Using the portion of the `volum` process interface shown in figure 3 the ATLAS code generator makes automatically the files `atl_volum.hh`, `atl_volum.H` and `atl_volum.C` which are depicted below.

`atl_volum.hh`

```
#ifndef __ATL_volumhh__
#define __ATL_volumhh__
#ifndef NOHEADER
#include "volum.h"
#endif
#include "atl_volum.H"
void segmentation(scene &,property);
void display_scene(scene);
simplex contained_in(point);
#endif
```

File `atl_volum.H` shows the code generated to define the *C++* prototypes for the external routines declared in the interface. This file also includes the `volum.h` file implemented by the developer because the prototypes use process types only known by the developer code.

atl_volum.H

```
#ifndef __ATL_volumH__
#define __ATL_volumH__
#include "Variable.H"
#include "atl_se.H"

namespace volum {
struct atl_point {float x; float y; float z;
atl_point() {}
atl_point(Variable &v) {
   if (v.Arbre()==NULL) atl_exit(-1); // Invalid variable
   x = ((nodereal *)(*(v.Arbre())).accedir(0))->Getvalor();
   y = ((nodereal *)(*(v.Arbre())).accedir(1))->Getvalor();
   z = ((nodereal *)(*(v.Arbre())).accedir(2))->Getvalor();
   }
operator Variable() {
   Type t("volum::point","S(x real,y real,z real)");
   Variable v(t,"");
   v.crea_arbre();
   *((*(v.Arbre())).accedir(0)) = x;
   *((*(v.Arbre())).accedir(1)) = y;
   *((*(v.Arbre())).accedir(2)) = z;
   return (v);
   }
   };
}
namespace volum {
struct atl_simplex {String name; atl_point vertices[4];
atl_simplex() {}
atl_simplex(Variable &v) {
   if (v.Arbre()==NULL) atl_exit(-1); // Invalid variable
   name = ((nodestring *) (*(v.Arbre())).accedir(0))->Getvalor();
   for (int i2=0;i2<4;i2++) {
      Variable v2("S(x real,y real,z real)",""); v2.crea_arbre();
      *(v2.Arbre())=*((*((*(v.Arbre())).accedir(1))).accedir(i2));
      atl_point tpaux(v2); vertices[i2]=tpaux;
      }
   }
operator Variable() {
   Type t("volum::simplex","S(name string,vertices V[4]S(x real,y real,z real))");
   Variable v(t,"");
   v.crea_arbre();
   *((*(v.Arbre())).accedir(0)) = name;
   for (int i2=0;i2<4;i2++) {
      *((*((*(v.Arbre())).accedir(1))).accedir(i2)) = *(((Variable) vertices[i2]).Arbre());
      }
   return (v);
   }
   };
}
namespace volum {
struct atl_scene{
   atl_simplex cont[100];
operator Variable() {
   Type t("volum::scene","V[100]S(name string,
                                  sides V[4]V[3]S(p1 S(x real,y real,z real),
                                                  p2 S(x real,y real,z real),
                                                  id integer))");
   Variable v(t,"");
   v.crea_arbre();
   for (int i1=0;i1<100;i1++) {
      *((*(v.Arbre())).accedir(i1)) = *(((Variable) cont[i1]).Arbre());
      }
   return (v);
   }
atl_scene() {}
atl_scene(Variable &v) {
   if (v.Arbre()==NULL) atl_exit(-1); // Invalid variable
   for (int i1=0;i1<100;i1++) {
      Variable v2("S(name string,
                    sides V[4]V[3]S(p1 S(x real,y real,z real),
                                    p2 S(x real,y real,z real),
                                    id integer))","");
```

```
      v2.crea_arbre(); *(v2.Arbre())=*((*(v.Arbre())).accedir(i1));
      atl_simplex tpaux(v2); cont[i1]=tpaux;
      }
   }
   };
}

using volum::atl_point;
using volum::atl_simplex;
using volum::atl_scene;
#endif
```

In file `atl_volum.H` we can see the *bridge type* definitions corresponding to the exported types defined in the interface: these are `atl_point`, `atl_simplex` and `atl_scene`. Their methods show how these *bridge types* are made from an ATLAS *Variable* and backwards.

`atl_volum.C`

```
#pragma implementation "taula.h"
#pragma implementation "Map.h"
#pragma implementation "VHMap.h"
#include "globals.H"
#include "ComunicDistr.H"
#include "Driver.H"
#include "String.h"
#include "gestio_crides.H"
#include "Variable.H"
#include "DLList.h"
#include "inc_atlas.H"

Comunic_Distr distrib(CANAL_COMUNIC_DISTR);
String nomprogram;
gestio_crida_a_rutina gestor_crides_ext;
Driver driv(distrib);

#ifndef NOHEADER
#include "volum.h"
#endif
#include "atl_volum.H"
#include "atl_volum.hh"

void aux_segmentation(String codi,DLList<Variable *> &parametres) {
   Pix p=parametres.first();
   atl_scene ptp0(*(parametres(p)));
   scene par0(ptp0);
   parametres.next(p);
   property ptp1(((nodeenter *)parametres(p)->Arbre())->Getvalor());
   parametres.next(p);
-->  segmentation(par0,ptp1);
   ptp0=par0.conversio_a_tipus_pont();
   Variable *rp0=new Variable(ptp0);
   ReturnParam *retpar0=new ReturnParam(codi,rp0);
   distrib.envia(retpar0);
   }
void aux_display_scene(String codi,DLList<Variable *> &parametres) {
   Pix p=parametres.first();
   atl_scene ptp0(*(parametres(p)));
   scene par0(ptp0);
   parametres.next(p);
-->  display_scene(par0);
   ReturnVoid *rv=new ReturnVoid(codi);
   distrib.envia(rv);
   }
void aux_contained_in(String codi,DLList<Variable *> &parametres) {
   Pix p=parametres.first();
   atl_point ptp0(*(parametres(p)));
   point par0(ptp0);
   parametres.next(p);
```

```
-->  simplex res=contained_in(par0);
     atl_simplex restp;
     restp=res.conversio_a_tipus_pont();
     Variable *vr=new Variable(restp);
     ReturnValue *rv=new ReturnValue(codi,vr);
     distrib.envia(rv);
     }
   void ini_per_crides() {
     gestor_crides_ext.lligar_nom_crida("segmentation",&aux_segmentation);
     gestor_crides_ext.lligar_nom_crida("display_scene",&aux_display_scene);
     gestor_crides_ext.lligar_nom_crida("contained_in",&aux_contained_in);
     }
   void main(int argc,char **argv) {
     nomprogram=argv[0];
     distrib.initialize(argv[1]);
     ini_per_crides();
     driv.set_name_program(nomprogram);
     ini_process();            // inicializations of the process itself
     driv.Dispatch();          // loop
     close(CANAL_COMUNIC_DISTR);
     exit(0);
     }
```

The last one, file `atl_volum.C`, shows the main code of the driver. This code includes the auxiliary routines for each external routine in the interface, and the main routine of the communications driver. The arrows have been added pointing to the places where the process routines are actually invoked.

5.2 Process code for translating to and from *bridge types*

The process code necessary for the types defined in this example depends on what the developer wants to add to the exported structure. The following code is the most common and easy class structure for the given types:

```
class point {
    float x,y,z;
  public:
    point (float a, float b, float c): x(a), y(b), z(c) {}
    point (atl_point p) { x = p.x; y = p.y; z = p.z; }
    atl_point conversio_a_tipus_pont ()
       { atl_point p; p.x = x; p.y = y; p.z = z; return p; }
    point &operator = (const point &p)
       { x = p.x; y = p.y; z = p.z; }
    float pos_x () { return x; }
    float pos_y () { return y; }
    float pos_z () { return z; }
};
class simplex {
    String name;
    point vertices[4];
  public:
    simplex (String n, point *vp): name(n)
       { for (int i=0; i<4; i++) vertices[i] = vp[i]; }
    simplex (atl_simplex s): name(s.name)
       { for (int i=0; i<4; i++) vertices[i] = s.vertices[i]; }
    atl_simplex conversio_a_tipus_pont ()
       { atl_simplex s; s.name = name;
         for (int i=0; i<4; i++)
           s.vertices[i] = vertices[i].conversio_a_tipus_pont ();
         return s; }
    String Name () { return name; }
    point *Vertices () const { return vertices; }
};
class scene {
    simplex cont[100];
  public:
```

```
    scene (simplex *vs)
       { for (int i=0; i<100; i++) cont[i] = vs[i]; }
    scene (atl_scene sc)
       { for (int i=0; i<100; i++) cont[i] = sc.cont[i]; }
    atl_scene conversio_a_tipus_pont ()
       { atl_scene sc;
         for (int i=0; i<100; i++)
           sc.cont[i] = cont[i].conversio_a_tipus_pont ();
         return sc; }
    simplex Cont (int i) { return cont[i]; }
};
```

Acknowledgements

This research has been partially supported by grants TIC-92-0605 and TIC-95-0630-C05-01 of the CICYT.

The first author has been supported by grant 1996FI-3100PG of the *Generalitat de Catalunya*.

References

[1] Jon Siegel. *CORBA Fundamentals and Programming*. OMG, 1996.

[2] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications*, 14(2), 1997.

[3] Gregory R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1), March 1991.

[4] Antoni Soto, Sebastià Vila, and Àlvar Vinacua. A Toolkit for constructing command driven graphics programs. *Computer & Graphics*, 16(4):375–382, 1992.

[5] Douglas C. Schmidt. The ADAPTIVE communication environment: Object-oriented network programming components for developing client/server applications. In *12th Sun Users Group Conference*, 1994.

[6] Douglas C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *Proceedings of the 1st Pattern Languages of Programs Conference*, August 1994.

[7] Marta Fairén and Àlvar Vinacua. ATLAS. Sistema de Comandes: Manual tècnic (in Catalan). *Report LSI-95-11-T*, 1995. http://www.lsi.upc.es/~mfairen.

[8] R. Srinivasan. Rfc 1832: Xdr: External data representation standard, August 1995.

[9] Marta Fairén and Àlvar Vinacua. Interprocess data transfer in ATLAS, a platform for distributed applications. *Report LSI-97-49-R*, 1997.